

CLEOPATRA

Building Responsive Systems from Physically-correct Specifications

AZER BESTAVROS*

Department of Computer Science
Boston University
Boston, MA 02215

Abstract

Predictability – the ability to foretell that an implementation will not violate a set of specified reliability and timeliness requirements – is a crucial, highly desirable property of responsive embedded systems. This paper overviews a development methodology for responsive systems, which enhances predictability by eliminating potential hazards resulting from physically-unsound specifications.

The backbone of our methodology is the Time-constrained Reactive Automaton (TRA) formalism, which adopts a fundamental notion of space and time that restricts expressiveness in a way that allows the specification of only reactive, spontaneous, and causal computation. Using the TRA model, unrealistic systems – possessing properties such as clairvoyance, caprice, infinite capacity, or perfect timing – cannot even be specified. We argue that this “ounce of prevention” at the specification level is likely to spare a lot of time and energy in the development cycle of responsive systems – not to mention the elimination of potential hazards that would have gone, otherwise, unnoticed.

The TRA model is presented to system developers through the *CLEOPATRA* programming language. *CLEOPATRA* features a C-like imperative syntax for the description of computation, which makes it easier to incorporate in applications already using C. It is event-driven, and thus appropriate for embedded process control applications. It is object-oriented and compositional, thus advocating modularity and reusability. *CLEOPATRA* is semantically sound; its objects can be transformed, mechanically and unambiguously, into formal TRA automata for verification purposes, which can be pursued using model-checking or theorem proving techniques. Since 1989, an ancestor of *CLEOPATRA* has been in use as a specification and simulation language for embedded time-critical robotic processes.

*This research was partially conducted while the author was at Harvard University and was partially supported by DARPA N00039-88-C-0163.

Report Documentation Page			<i>Form Approved OMB No. 0704-0188</i>	
<p>Public reporting burden for the collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to a penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number.</p>				
1. REPORT DATE 1993	2. REPORT TYPE	3. DATES COVERED 00-00-1993 to 00-00-1993		
4. TITLE AND SUBTITLE CLEOPATRA. Building Responsive Systems from Physically-correct Specifications			5a. CONTRACT NUMBER	
			5b. GRANT NUMBER	
			5c. PROGRAM ELEMENT NUMBER	
6. AUTHOR(S)			5d. PROJECT NUMBER	
			5e. TASK NUMBER	
			5f. WORK UNIT NUMBER	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Defense Advanced Research Projects Agency, 3701 North Fairfax Drive, Arlington, VA, 22203-1714			8. PERFORMING ORGANIZATION REPORT NUMBER	
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)			10. SPONSOR/MONITOR'S ACRONYM(S)	
			11. SPONSOR/MONITOR'S REPORT NUMBER(S)	
12. DISTRIBUTION/AVAILABILITY STATEMENT Approved for public release; distribution unlimited				
13. SUPPLEMENTARY NOTES				
14. ABSTRACT see report				
15. SUBJECT TERMS				
16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES 26
a. REPORT unclassified	b. ABSTRACT unclassified	c. THIS PAGE unclassified		

1 Introduction

A computing system is *embedded* if it is a component of a larger system whose primary purpose is to monitor and control an environment. The leaping advances in computing technologies that the last few decades have witnessed have resulted in an explosion in the extent and variety of such systems. This trend is expected to continue in the future.

Embedded systems are usually associated with critical applications, in which human lives or expensive machinery are at stake. Their missions are often long-lived and uninterruptible, making maintenance or reconfiguration difficult. Examples include command and control systems, nuclear reactors, process-control plants, robotics, avionics, switching circuits and telephony, data-acquisition systems, and real-time databases, just to name a few. The sustained demands of the environments in which such systems operate pose relatively rigid and urgent performance requirements. These requirements are usually stated as timing constraints on their behaviors. Wirth [Wirt77] singled out this processing-time dependency as the one aspect that differentiates real-time from other sequential and parallel systems. This led to a body of research on *real-time computing*, which encompasses issues of specification techniques, validation and prototyping, formal verification, safety analysis, programming languages, development tools, scheduling, and operating systems. In addition to timeliness, embedded systems are also required to meet stringent reliability constraints, which are usually stated as behavioral safety and liveness invariants. For comprehensive surveys of recent research in real-time systems, the reader is directed to [Stan88b, Burn90, Tilb91a, Tilb91b].

The absence of a unified suitable formal framework that addresses the aforementioned issues severely limits the usefulness of these studies. This situation is further exacerbated considering the range of disciplines employed in developing the various components of an embedded application. For example, in a simple sensori-motor robotic application [Clar91], algorithms from various disciplines like low-level imaging, active vision, tactile sensing, path planning, compliant motion control, and non-linear dynamics may be utilized [Fu87]. Not only are these disciplines different in their abstractions and programming styles, but also they differ in their computational requirements, which range from single-board dedicated processors to massively parallel general-purpose computers.

Current embedded systems are expensive to build and their properties are verified with ad hoc techniques[Stan88a]. Schneider [Schn88] portrays the situation aptly by saying that “Unlike other engineering disciplines, our methods are not founded on science. Real-time systems are built one way or another because that was the way the ‘last one’ was built. And, since the ‘last one’

worked, we hope that the next one will”. This brute force approach is not likely to scale-up with future systems.

In this paper we propose *CLEOPATRA*,¹ a programming environment that recognizes the unique requirements of responsive embedded systems. *CLEOPATRA* features a C-like imperative syntax for the description of computation, which makes it easier to incorporate in applications already using C. It is event-driven, and thus appropriate for embedded process control applications. In particular, rather than describing behaviors using *control* structures, it describes behaviors using time-constrained causal structures. *CLEOPATRA* is object-oriented and compositional, thus advocating modularity and reusability. *CLEOPATRA* is semantically sound; its objects can be transformed, mechanically and unambiguously, into formal automata for verification purposes. Our experience with *CLEOPATRA* confirms its suitability as a vehicle for the specification and validation of many embedded and time-critical applications. In particular, we used it to simulate and analyze asynchronous digital circuits, sensori-motor behavior of autonomous creatures, and intelligent controllers [Best91a, Best90c, Best90b]. A compiler that allows the execution of *CLEOPATRA* specifications has been developed [Best92], and is available via FTP from `cs.bu.edu:/bestavros/cleopatra/`.

CLEOPATRA is based on the Time-constrained Reactive Automata (**TRA**) formalism [Best91b, Best91c]. Using the **TRA** formalism, an embedded system is viewed as a set of asynchronously interacting automata (**TRAs**), each representing an autonomous system entity. **TRAs** are reactive in that they abide by Lynch’s input enabling property [Lync88b]; they communicate by signaling events on their output channels and by reacting to events signaled on their input channels. The behavior of a **TRA** is governed by time-constrained causal relationships between computation-triggering events. The **TRA** model is compositional and allows only benign time, control, and computation non-determinism. Using the **TRA** formalism, there is no conceptual distinction between a system and a property; both are specified as formal objects. This reduces the verification process to that of establishing correspondences – preservation and implementation – between such objects.

This paper is organized as follows. In Section 2, we overview the **TRA** model and highlight its suitability for the specification of embedded systems. In our overview, we emphasize the **TRA** operational semantics, which underlies the execution model of *CLEOPATRA*. In Section 3, we describe the *CLEOPATRA* specification/programming language. In Section 4, we present a compiler that allows the execution of *CLEOPATRA* specifications. In Section 5, we conclude with current and future research directions.

¹A *C*-based *L*anguage for the *E*vent-driven *O*bject-oriented *P*rototyping of *A*synchronous *T*ime-constrained *R*eactive *A*utomata.

2 The TRA Model

The TRA model has evolved from our earlier work in [Best90a] extending Lynch's IOA model [Lync88b, Lync88a] to suit embedded and time-constrained computation.

2.1 Novelties

Previous studies in modeling real-time computing have focussed on adding the notion of time without regard to physical properties of the modeled systems. This makes it possible to specify systems that do not abide by principles like causality and spontaneity. Using the TRA model, requirements that are physically impossible to guarantee are not possible to express. This preventative approach is likely to spare a lot of time and energy in the development cycle (specification, implementation, and verification) of responsive systems.

The TRA model deals not only with the notion of time, but also with the notion of space. Events occur at uniquely identifiable points in time as well as in state space. Concurrent events are permitted only if they affect disjoint state subspaces. The payoff for this dual treatment of space and time is manifold. In particular, mappings between various levels of abstractions for compilation and verification purposes become more robust as the formalism becomes more structured.

The TRA model does not allow the specification of systems that are not *reactive*. A system is reactive if it cannot block the occurrence of events not under its control. This property is crucial for accurate and realistic modeling of embedded and real-time systems. A sufficient condition for reactivity is the *input enabling* property proposed in [Lync88b]. The TRA model is input enabled. It distinguishes clearly between environment-controlled actions, which cannot be restricted or constrained, and locally-controlled actions, which can be scheduled and disabled.

A non-deterministic system is *causal* if given two inputs that are identical up to any given point in time, there exist outputs (for the respective inputs) that are also identical up to the same point in time. The TRA model enforces causality by requiring that any locally-controlled actions be produced only as a *result* of an earlier *cause*. In our work, a clear distinction is made between causality and dependency. An event occurs as a result of exactly one earlier event but may depend on many others as reflected in the state of the system. This spares our formalism from dealing with clairvoyant and capricious behaviors [Stua91].

Spontaneity is a notion closely related to causality.² A system is *spontaneous* if its output

²Actually both spontaneity and causality are directly related to the past and future light cones of an event in space-time [Hawk88].

actions at any given point in time t cannot depend on actions occurring at or after time t . In particular, if an output occurs simultaneously with (say) an input transition, the same output could have been produced without the simultaneous input transition [Sree90]. Simultaneity is, thus, a mere coincidence; the output event could have occurred spontaneously even if the input transition was delayed. The **TRA** model enforces spontaneity by requiring that simultaneously occurring events be independent; time has to *necessarily* advance to observe dependencies.

The **TRA** model distinguishes between two notions of time: *real* and *perceived*. Real time cannot be measured by any single process in a given system; it is only observable by the environment. Perceived time, on the other hand, can be specified using uncertain time delays. The **TRA** model, therefore, does not provide for (or allow the specification of) any *global* or *perfect clocks*. As a consequence, the only measure of time available for system processes has to be relative to *imperfect, local clocks*. This distinction between real time and perceived time is important when dealing with embedded applications where time properties are stated with respect to real time, but have to be preserved relying on perceived time.

2.2 Basic definitions

We adopt a continuous model of time similar to that used in [Alur90, Lewi90]. We represent any point in time by a nonnegative real $t \in \mathbb{R}$. Time intervals are defined by specifying their end-points which are drawn from the set of nonnegative rationals $\mathcal{Q} \subset \mathbb{R}$. A time interval is viewed as a traditional set over nonnegative real numbers. It can be an empty set, in which case it is denoted by ε , it can be a singleton set, in which case it is denoted by the $[t, t]$, $t \in \mathcal{Q}$, or else it can be an infinite set, in which case it is denoted by $[t_l, t_u]$, $(t_l, t_u]$, $[t_l, t_u)$, or (t_l, t_u) – the right-closed, left-closed, and open time intervals, respectively, where $t_l, t_u \in \mathcal{Q}$ and $t_l < t_u$. The set of all such infinite time intervals is denoted by \mathcal{D} .

A real-time system is viewed as a set of interacting mealy automata called **TRAs**. **TRAs** communicate with each other through *channels*. A channel is an abstraction for an *ideal* unidirectional communication. The information that a channel carries is called a *signal*, which consists of a sequence of *events*. An event underscores the occurrence of an *action* at a specific point in time. An action is a *value* associated with a channel. For example, let **North**, **South**, **East**, and **West** be the possible values that can be signaled on some channel **MOVE** of a given **TRA**. **MOVE(East)** is, therefore, a possible action of the **TRA**. The instantiation of **MOVE(East)** at time t_1 denotes the occurrence of an event $\langle \text{MOVE(East)} : t_1 \rangle$. The sequence of events $\langle \text{MOVE(East)} : t_1 \rangle \langle \text{MOVE(North)} : t_2 \rangle \langle \text{MOVE(South)} : t_3 \rangle \dots$ etc. constitutes a signal. Signals are single valued; they cannot convey more than one event simultaneously. That is, for a signal $\langle a_0 : t_0 \rangle \langle a_1 : t_1 \rangle \dots \langle a_k : t_k \rangle \dots$ we require that $t_k < t_{k+1}$, $k \geq 0$.

At any point in time, a **TRA** is in a given *state*. The set of all such possible states defines the **TRA**'s *state space*. The state of a **TRA** is visible and can only be changed by local *computations*. Computations (and thus state transitions) are triggered by actions and might be required to meet specific timing constraints.

2.3 TRA Objects

Definition 1 A **TRA** object is a sextuple $(\Sigma, \sigma_0, \Pi, \Theta, \Lambda, \Upsilon)$, where:

- ◊ Σ , the **TRA** signature, is the set of all the **TRA** channels. It is partitioned into three disjoint sets of input, output, and internal channels, denoted by Σ_{in} , Σ_{out} , and Σ_{int} , respectively. The set consisting of both input and output channels is the set of external channels (Σ_{ext}). These are the only channels visible from outside the **TRA**. The set consisting of both output and internal channels is the set of local channels (Σ_{loc}). These are the locally controlled channels of the **TRA**.
- ◊ $\sigma_0 \in \Sigma_{\text{in}}$ is the start channel.
- ◊ Π , the signaling range function, maps each channel in Σ to a possibly infinite set of values that can be signaled as actions on that channel. Action sets of different channels are disjoint. The set of all the actions of a **TRA** is given by $\Pi(\Sigma)$. The set of input, output, internal, external, and local actions are similarly given by $\Pi(\Sigma_{\text{in}})$, $\Pi(\Sigma_{\text{out}})$, $\Pi(\Sigma_{\text{int}})$, $\Pi(\Sigma_{\text{ext}})$, and $\Pi(\Sigma_{\text{loc}})$, respectively.
- ◊ Θ is a possibly infinite set of **TRA** states. The set Θ is expressed as the cross product of a finite number of subspaces $\Theta = \Phi_1 \times \Phi_2 \times \dots \times \Phi_p$, where $p \geq 1$ is the dimension of the state space.
- ◊ $\Lambda \subseteq \Theta \times \Pi(\Sigma) \times \Theta$ is a set of possible computational steps of the **TRA**. **TRAs** are input enabled which means that for every $\pi \in \Pi(\Sigma_{\text{in}})$, and for every $\theta \in \Theta$, there exists at least one step $(\theta, \pi, \theta') \in \Lambda$, for some $\theta' \in \Theta$. Thus, Λ defines a total multifunction $\Lambda : \theta \times \Pi(\Sigma_{\text{in}}) \rightarrow \theta$.
- ◊ $\Upsilon \subseteq \Sigma \times \Sigma_{\text{loc}} \times \mathcal{D} \times 2^\Theta$ is a set of time-constrained causal relationships (or simply time constraints) of the **TRA**. A time constraint $v_i \in \Upsilon$ is a quadruple $(\sigma_i, \sigma'_i, \delta_i, \Theta_i)$ whose interpretation is that: if an action is signaled at time $t \in \mathbb{R}$ on the channel σ_i , then a corresponding action must be fired on the channel σ'_i at time t' , where $t' - t \in \delta_i$, provided that the **TRA** does not enter any of the states in Θ_i for the open interval (t, t') .³ The channel $\sigma_i \in \Sigma$ is called the trigger of the time constraint, whereas $\sigma'_i \in \Sigma_{\text{loc}}$ is called the constrained channel. $\Theta_i \subseteq \Theta$ defines the set of states that disable the time constraint; once triggered a time constraint becomes and remains active until satisfied or disabled. A time constraint is satisfied by the firing of an action on the channel σ_i within the imposed time bounds; it is disabled if the **TRA** enters in one of the disabling states in Θ_i before it is satisfied. The interval δ_i specifies upper and lower bounds on the delay between the triggering and satisfaction (or disabling) of the time constraint v_i .

³Notice that this condition does not necessitate the existence of a computational step $(\theta, \pi', \theta') \in \Lambda$ for each $\theta \in \Theta - \Theta_i$, where $\pi' \in \Pi(\sigma')$ and $\theta' \in \Theta$, since the specification of the **TRA** might avoid being in θ when σ' is scheduled to fire.

As an example of a TRA specification, consider the the up/down counter whose state diagram is shown in Figure 1. The counter accepts commands issued on the input channel `cmd` to count up or down and signals the value of the current count (state) on the output channel `cnt`. The counter starts its operation once an action is fired on the `init` channel. The value of the `init` signal determines the starting state of the counter. The counter is constrained to produce a count every at least 1.9 and at most 2.1 units of time, once it starts execution. Figure 1 shows the TRA-specification of such a counter.

The first three components of a **TRA** sextuple can be viewed as defining an interface between the **TRA** object and its environment. In particular, to be able to use the counter of Figure 1, it suffices to know its external signature $\Sigma_{\text{in}} = \{\text{init}, \text{cmd}\}$, $\Sigma_{\text{out}} = \{\text{cnt}\}$, the identity of the start channel $\sigma_0 = \text{init}$, along with the signaling range of all the channels in Σ_{ext} . The last three components of a **TRA** sextuple are responsible for its behavior. The state space defines the spatial structure of the computation. For the counter of Figure 1, this structure is unidimensionally spanned by the single state variable θ . The set of computational steps defines the effect of events on the state of the **TRA**. The set of time-constrained causalities defines the rules governing the *scheduling* of the **TRA**'s local events. For the counter of Figure 1, there are two such rules.

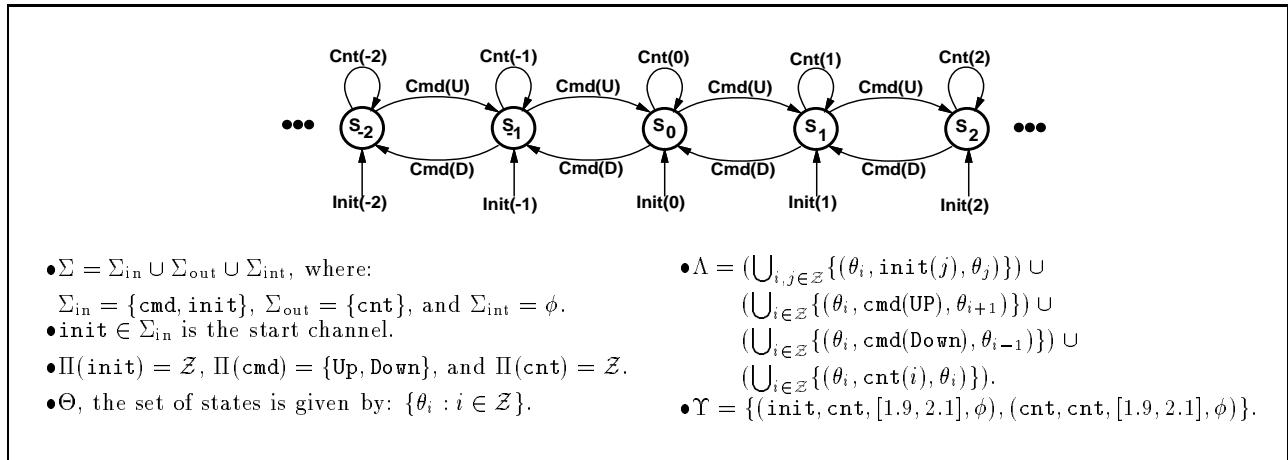


Figure 1: TRA-specification of up/down counter.

2.4 Space and Time aspects of TRAs

The behavior of a **TRA** is generally non-deterministic. Three sources of non-determinism can be singled out. In a given state there might be a number of choices concerning the action to be fired. Each one of these choices results in a different computational step, and thus in a different execution. This gives rise to *control* non-determinism. The **TRA** timing constraints specify lower and upper bounds on the delay between causes and effects, thus leaving the **TRA** with a potentially infinite

number of choices concerning the exact delay to be exhibited. Each one of these choices results in a different event, and thus in a different execution. This gives rise to *timing* nondeterminism. Finally, the computation associated with specific actions might be non-deterministic. In this case, firing the same action from the same state might result in different next states, and thus in different executions. This gives rise to *computation* non-determinism. Considered separately, each one of the above forms of non-determinism is benign. A combination thereof, however, deserves a closer attention. In particular, the interplay between control non-determinism and timing non-determinism is interesting because it is related to the notions of space and time. Control non-determinism refers to uncertainties about the identity of the channel that will be fired; it refers to a *spacial uncertainty*. As such, and to abide by the spontaneity principle, it must reduce the range of possible *timing uncertainty*.

To illustrate this point, consider a **TRA**, \mathcal{A} , for which two possible steps are: $(\theta_i, \pi_1, \theta_j)$ and $(\theta_i, \pi_2, \theta_k)$, where $\theta_j \neq \theta_k$. Furthermore, assume that \mathcal{A} entered state θ_i at time t and that both π_1 and π_2 are scheduled. Now, if the timing constraints for π_1 and π_2 are specified such that both actions can fire on different channels at some later time t' , then “what will be the next state of \mathcal{A} ? Will it be θ_j or θ_k or neither?”⁴ The issue here is not whether the next state should be θ_j or θ_k . Rather, the issue is whether or not such a situation should have been allowed in the first place.

Two computational steps *conflict* if both of them introduce changes to at least one of the subspaces of the **TRA**’s state space. This is formally defined below.

Definition 2 *Two computational steps $(\theta_i, \pi_i, \theta'_i), (\theta_j, \pi_j, \theta'_j) \in \Lambda$ conflict if and only if for some dimension k of Θ , $\theta_i[k] \neq \theta'_i[k]$ and $\theta_j[k] \neq \theta'_j[k]$, where $1 \leq k \leq n$.*

It is important to realize that the conflict relationship depends not only on a **TRA**’s computational behavior, but also on the structure of its state space. In particular, two **TRAs** with isomorphic computational steps could have very different conflict relationships depending on their state space characterizations. The notion of conflicting computational steps can be easily extended to actions and channels.

The conflict relationship depicts computational dependencies that emerge due to sharing information about state. For two local actions to conflict, their respective channels must be under the control of a single *component* of the **TRA**. The transitive closure of the conflict relationship, therefore, defines a partition on the locally-controlled channels of a given **TRA**.

Definition 3 *Two local channels σ_i and σ_j belongs to the same component (class) if they conflict.*

⁴The argument given here is made assuming that both π_1 and π_2 are locally-controlled actions. The same argument, however, can be made if either π_1 or π_2 , or both are input actions.

The partition into classes of the **TRA**'s locally-controlled channels captures some of the structure of the system the automaton is modeling or the set of requirements it is specifying. In particular, each class of channels is intended to represent the set of channels locally-controlled by *some* system component. This partitioning retains the basic control structure of the system's primitive components and provides a concrete notion of spacial locality.

The actions on the input channels of a given **TRA** are not under its control; they can fire at any time. To preserve the non-blocking (input-enabled) nature of the **TRA** model, it is, therefore, necessary to insure that input actions on different channels do not conflict. A **TRA** is improper if at least two of its input channels conflict, otherwise it is proper. For the remainder of this paper, it will be assumed that any **TRA** is *proper* unless otherwise stated.

The notion of system components we are presenting here is novel and entirely different from that used in untimed models to express fairness [Lync88b] by requiring that, in an infinite execution, each of the system's components gets infinitely many chances to perform its locally-controlled actions. In timed systems, the major concern is *safe* and not necessarily *fair* executions [Schn88]. Even if required, fairness can be enforced by treating it as a safety property; liveness properties can be handled in infinite execution by requiring time to grow unboundedly.⁵. This led to the abandoning of the idea of partitioning a system into components in our earlier model proposed in [Best90a]. Lynch and Vaandrager [Lync91] followed suit in their recent modification of the model proposed in [Tutt88]. In the **TRA** model we use system components to represent what can be termed as *spacial locality*. Different actions can be signaled at the same “time” only if they are not signaled from the same “place”; they can be produced at the same “place” only if they do not occur at the same “time”.⁶

2.5 TRA Executions and Behaviors

In standard automata theory, there is no distinction between choosing a transition and firing it; they constitute a unique, instantaneous, and atomic activity. In the **TRA** model a distinction is made whereby choosing (scheduling) a transition and executing (committing) that transition are separate activities. They are *distinct* in that they are separated in time. In fact, a scheduled transition does not have to be committed; it can be abandoned due to unforeseeable conditions. The distinction between the two activities is also pronounced in the way the **TRA** model differentiates between input and local events. Input events are not under the **TRA**'s control; they cannot be blocked or delayed. Local events are under the **TRA**'s control; they are time constrained, and could be disabled.

⁵Such executions were called *admissible* in [Lync91]

⁶This intuition is inspired from physical systems, where events are characterized and distinguishable by their time-space coordinates [Hawk88].

Consider the time constraint $v_i = (\sigma_i, \sigma'_i, \delta_i, \Theta_i) \in \Upsilon$. v_i identifies a time-constrained causal relationship between the events signaled on σ_i and those signaled on σ'_i . In particular, the occurrence of a triggering event on σ_i results in an intention to perform an action on σ'_i within the time frame imposed by δ_i . The commitment or abandonment of such an intention in due time is conditional on the states assumed by the **TRA** from when the intention is posted until when it is committed or abandoned. At any given point in time, a **TRA** might have several outstanding intentions. In particular, the occurrence of a single event might generate a number of intentions, each dictated by a different time constraint. Different outstanding intentions are not necessarily imposed by different time constraints. In particular, the repeated occurrence of a triggering event might generate a number of outstanding intentions, all of which are posted by the same time constraint.

The *state* of a **TRA** at an arbitrary point in time is not sufficient to construct its *future behavior*. In addition to the state, the intervals of time where scheduled transitions might fire (due to earlier triggers) have to be recorded. For a given **TRA**, we define the *intention vector* $I = \vec{\Delta}$ to be a vector of r sets of intentions, where $r = |\Upsilon|$. Each entry in I is associated with one of the **TRA**'s time constraints. In particular, if $v_i = (\sigma_i, \sigma'_i, \delta_i, \Theta_i) \in \Upsilon$ is one of the **TRA**'s time constraints, then $I[v_i] = \{\delta_{i1}, \delta_{i2}, \dots, \delta_{ik}, \dots, \delta_{im}\}$ denotes a set of m time intervals during which actions on the channel σ'_i are intended to be fired as a result of earlier triggers on σ_i . Each one of the intervals in Δ_i can be thought of as an independent *activation* of the time constraint v_i . An empty intentions set, $I[v_i] = \phi$, indicates the absence of any activations of v_i . The empty intention vector, I_ϕ , consists of r such empty sets.

Definition 4 *We define the status of a **TRA** at any point in time $t \in \mathfrak{R}$ to be the tuple (θ, I) , where θ and I are the **TRA**'s state and intention vector at time t , respectively.*

A **TRA** changes its status only as a response to the occurrence of an input or an intended local event. In other words, the change in a **TRA**'s status is necessarily a causal *reaction* to an input event or to an earlier triggering event. Assume that the status (θ, I) of a **TRA** was entered at time t as a result of an event $\langle \pi : t \rangle$, where $\pi \in \Pi(\sigma_j), \sigma'_j \in \Sigma$. Furthermore, assume that at time t' ($t' \geq t$), an action $\pi' \in \Pi(\sigma'_j)$ is fired, where $\sigma'_j \in \Sigma$. As a result, the **TRA** will assume a new status (θ', I') . The status (θ', I') is called a *successor* of the status (θ, I) due to the event $\langle \pi' : t' \rangle$. Five conditions – namely, legality, spontaneity, safety, causality, and consistency – have to be met for such a succession to occur.

Definition 5 *Assume that the status (θ, I) of a **TRA** was entered at time t . Furthermore, assume that at a later time $t' > t$, a set of simultaneous actions $\pi_1 \in \Pi(\sigma_1), \pi_2 \in \Pi(\sigma_2), \dots, \pi_m \in \Pi(\sigma_m)$ were fired, where $\sigma_j \in \Sigma, 0 \leq j \leq m$. As a result, the **TRA** will assume a new status (θ', I') , where $I' = (I \cup I'_{\text{enabled}}) - (I'_{\text{fired}} \cup I'_{\text{disabled}})$.*

The status (θ', I') is called a valid successor of the status (θ, I) due to the occurrence of the set of simultaneous events $\langle \pi_1, \pi_2, \dots, \pi_m : t' \rangle$, if and only if the following conditions hold:

1. Spontaneity:

The channels $\sigma_1, \sigma_2, \dots, \sigma_m$ do not conflict; they belong to different TRA components.

2. Legality:

There exists some sequence of transitions $(\theta, \pi_1, \theta_1), (\theta, \pi_2, \theta_2), \dots, (\theta, \pi_m, \theta_m) \in \Lambda$, such that $\theta_m = \theta'$.

3. Safety:

For every intention $\delta_{ik} \in I[v_i]$, $t'' \in \delta_{ik}$ for some $t'' > t'$, $t'' \in \mathfrak{R}$, where $v_i \in \Upsilon$.

4. Causality:

For all $\sigma_i \in \Sigma_{\text{loc}}$, the following conditions hold

- a. If $\sigma_i \neq \sigma_j$ for all $1 \leq j \leq m$ then for every $v_k = (\sigma_k, \sigma'_k, \delta_k, \Theta_k) \in \Upsilon$ for which $\sigma'_k = \sigma_i$, $I'_{\text{fire}}[v_k] = \phi$.
- b. Otherwise, let $\Upsilon_i \subseteq \Upsilon$ be the set of time constraint with σ_i as the constrained channel, then there must exist exactly one time constraint $v_r \in \Upsilon_i$ such that:
 - ◊ $I'_{\text{fire}}[v_r] = \{\delta_{rk}\}$, where $\delta_{rk} \in I[v_r]$ and $t' \in \delta_{rk}$, and
 - ◊ $I'_{\text{fire}}[v_k] = \phi$, where $v_k \in \Upsilon_i$ and $v_k \neq v_r$.

5. Consistency:

For every time constraint $v_k = (\sigma_k, \sigma'_k, \delta_k, \Theta_k) \in \Upsilon$, the following conditions hold

- a. If $\theta' \in \Theta_k$, then
 - ◊ $I'_{\text{disabled}}[v_k] = I[v_k]$ and
 - ◊ $I'_{\text{enabled}}[v_k] = \phi$.
- b. Otherwise
 - ◊ $I'_{\text{disabled}}[v_k] = \phi$, and
 - ◊ If $\sigma_k = \sigma_j$ for some $1 \leq j \leq m$, then $I'_{\text{enabled}}[v_k] = \{(t' + \delta_i)\}$, else $I'_{\text{enabled}}[v_k] = \phi$.

In the above definition, the *spontaneity* condition allows the occurrence of simultaneous events only if they do not conflict. This guarantees that the transition from θ to θ' is independent of the ordering of concurrent computational steps. The *legality* condition ensures that the state change from θ to θ' is the result of defined computational steps. The *safety* condition guarantees that no active time constraint expires. In other words, outstanding intentions are either committed or abandoned *in due time*. The *causality* condition necessitates that local events be causal; they are signaled only if intended due to an earlier trigger. Thus, the causality condition guarantees that there is exactly one committed intention per local event. In other words, every local event satisfies exactly one intention. The *consistency* condition requires that the intentions in I continue to exist in I' unless otherwise dictated by the occurrence of the set of simultaneous events $\langle \pi_1 : t' \rangle \langle \pi_2 : t' \rangle \dots \langle \pi_m : t' \rangle$.

We use the notation $(\theta, I) \xrightarrow{\langle \pi_1, \pi_2, \dots, \pi_m : t' \rangle} (\theta', I')$ to denote the *direct status succession* from (θ, I) to (θ', I') due to the firing of the set of simultaneous events $\langle \pi_1 : t' \rangle, \langle \pi_2 : t' \rangle, \dots, \langle \pi_m : t' \rangle$. Furthermore, we use the notation $(\theta, I) \xrightarrow{\alpha} (\theta', I')$ to denote the *extended status succession* from (θ, I) to (θ', I') due to a number of direct status successions.

A **TRA** is said to have reached a *stable status* $(\hat{\theta}, \hat{I})$, if all entries of the intention vector are empty ($\hat{I} = I_\phi$). A **TRA** remains in a stable status until excited by an input event. This follows directly from the causality requirement for a status succession.

To start executing, a **TRA** $(\Sigma, \sigma_0, \Pi, \Theta, \Lambda, \Upsilon)$ is put in a stable *initial status* (θ_0, I_0) , where $I_0 = I_\phi$ and $\theta_0 \in \Theta$. The execution is initiated at time t_0 with the firing of an action π_0 on the start channel σ_0 , where $\pi_0 \in \Pi(\sigma_0)$. An execution e of a **TRA** is a possibly infinite string of alternating statuses and events, which starts with an initial status followed by an initiating event, and which contains an infinite number of status successions (infinite execution), or terminates in a stable status (finite execution).

We follow an approach similar to that adopted in [Lync88b] by defining β to be a *behavior* of a **TRA** \mathcal{A} , if it consists of all the *external* events appearing in some execution e of \mathcal{A} . We denote the set of all the possible behaviors of a **TRA** \mathcal{A} by $behs(\mathcal{A})$. Obviously, $behs(\mathcal{A})$ describes all the possible interactions that the **TRA** \mathcal{A} might be engaged in, and, therefore, constitutes a complete specification of the system that \mathcal{A} models.

A **TRA** \mathcal{A} is said to *implement* another **TRA** \mathcal{B} if \mathcal{A} does not produce any behavior that \mathcal{B} could have produced. In other words, all of \mathcal{A} 's behaviors (the implementation) are possible behaviors of \mathcal{B} (the specification). The reverse, however, is not true. There might exist behaviors of \mathcal{B} that cannot be generated by \mathcal{A} . The notion of a **TRA** implementing another is used mainly in verification.

2.6 TRA Composition

A basic aspect of the **TRA** model is its capability to model a complex system by operating on simpler system components. In this section we examine such an operation, namely composition. Other operations (for example hiding and renaming) were presented in [Best91c].

The composition of a countable collection of *compatible* **TRAs**, $\{\mathcal{A}_i : i \in \mathcal{I}\}$, is a new **TRA** $\mathcal{A} = \mathcal{A}_0 \times \mathcal{A}_1 \times \dots \times \mathcal{A}_i \times \dots = \Pi_{i \in \mathcal{I}} \mathcal{A}_i$. The execution of \mathcal{A} involves the execution of all its components $\mathcal{A}_{i \in \mathcal{I}}$, each starting from an initial status and observing every external event signaled by either the environment (input) or by any **TRA** in the collection $\{\mathcal{A}_i : i \in \mathcal{I}\}$. The *compatibility* condition for composition insures that, for each channel in the composition, there is at most one writer, a finite number of readers, and that the signaling ranges of readers and writers are compatible.

The input signature of the composed **TRA** consists of those channels that are inputs to one or more of the component **TRAs**, and which are not outputs of any of the component **TRAs**. The output signature of the composed **TRA** consists of all the outputs of all the component **TRAs**. Similarly, the internal signature of the composed **TRA** consists of all the internal channels of all the component **TRAs**. The start channel of the composed **TRA** is the start channel of one or more of its component **TRAs**.⁷ The signaling range function of the composed **TRA** is defined so as to preserve its input-enabled property. In particular, the signaling range of an input channel consists of only those actions that can be accepted by all readers of that channel. A computational step of the composed **TRA** is necessarily a step of one of its components. Similarly the time-constrained causal relationships of the composed **TRA** are exactly those of the component **TRAs**.

In [Best91c], the formal construction of the sextuple representation of a composition is given. Also, the relationships between the behaviors and spacial properties of the composed **TRA** and those of its constituent **TRAs** are established. In particular, we prove that the sets of proper, spontaneous, and causal **TRAs** are closed under composition.

The **TRA** composition operation is more general than those reported in [Lync88b, Tutt88, Best90a] in that it allows the specification of both *parallel* and *sequential* composition. In particular, the introduction of the *start channel* permits the execution of two **TRAs** to be concurrent if they share the same start channel, or to be serialized if the start channel of one (child) is an output of the other (parent). Through appropriate composition, our model is capable of representing all of the composition operations in [Lyon89].

3 *CLEOPATRA*: A **TRA**-based Specification Language

In *CLEOPATRA*, systems are specified as interconnections of **TRA** objects. Each **TRA** object has a set of *state variables* and a set of *channels*. Time-constrained causal relationships between events occurring on the different channels, and the computations (state transitions) that they trigger, are specified using *Time-constrained Event-driven Transactions* (TETs). The behavior of a **TRA** object is described using TETs. **TRA** objects can be composed together to specify more complex **TRAs**.

The correspondence between *CLEOPATRA* and the **TRA** formalism is straightforward. Every object in *CLEOPATRA* corresponds to a **TRA** sextuple. In [Best91c], the construction of a **TRA** sextuple, given a *CLEOPATRA* object, is detailed.

⁷Without loss of generality, we assume that **TRA** to be \mathcal{A}_0 .

3.1 Classes and Objects

A TRA object specification in *CLEOPATRA* consists of two components: a header and a body. An object's header specifies its name, the parameters needed for its instantiation, and its signature. An object's body specifies its behavior. In its simplest form, this entails the specification of the TRA's state space and its potentially time-constrained set of reactions to the different events visible to it. More complex behaviors include (among others) the specification of: internal channels, initialization code, and interconnection of local (composed) objects. Figure 2 shows a BNF-like description of a TRA object in *CLEOPATRA*.

In *CLEOPATRA*, TRAs are defined in *classes*. For example, Figure 3 shows the *CLEOPATRA* specification of the class of integrators that use trapezoidal approximation.

```

<tra-object> := <tra-header> '{' <tra-body> '}'
<tra-header> := 'TRA-class' <tra-name> {'(' <tra-params-spec> ')'} <signature>
<tra-params-spec> := {<type> <param-id> ';' <tra-params-spec>}
<signature> := {<ch-list-spec> '-' <ch-list-spec>}
<ch-list-spec> := <ch-id> ( <type> ) {',', <ch-list-spec>}
<type> := 'int' | 'double' | 'bool' | ...
<tra-body> := {<declarations>} {<init>} {<transactions>}
<declarations> := {<state>} {<internal>} {<included>}
<state> := 'state:' <state-var-def>
<state-var-def> := <type> <var-list-def> ';' {<statevar-def>}
<var-list-def> := <var-id> {'=' <constant-exp>} {',', <var-list-def>}
<internal> := 'internal:' <signature>
<included> := 'included:' <included-objects>
<included-objects> := <tra-instantiation> ';' {<included-objects>}
<tra-instantiation> := <tra-name> {'(' <actual-param-list> ')'} <ext-binding>
<actual-param-list> := <constant-exp> {',', <actual-param-list>}
<ext-binding> := {<ch-list>} '-' {<ch-list>}
<ch-list> := <ch-id> {',', <ch-list>}
<init> := <code>
<transactions> := {<xact> {<transactions>}}
<xact> := <xact-header> ':' <xact-body>
<xact-header> := {<trigger-list>} '-' <out-sig-spec>
<trigger-list> := <in-sig-spec> {',', <trigger-list>}
<in-sig-spec> := <ch-id> '(' {<var-id>} ')'
<out-sig-spec> := <ch-id> '(' {<exp>} ')'
<xact-body> := <act> | '{' <acts> '}'
<acts> := <act> {<acts>}
<act> := <computation> | {<condframe>} <fire-acts> | {<timeframe>} <fire-acts>
<computation> := 'commit' '{' <code> '}' | 'do' '{' <code> '}'
<condframe> := 'unless' '(' <cond> ')' | 'while' '(' <cond> ')'
<timeframe> := <closed-timeframe> | <open-timeframe>
<closed-timeframe> := 'within' '[' <constant-exp> '^' <constant-exp> ']'
<open-timeframe> := 'before' <constant-exp> | 'after' <constant-exp>

```

Figure 2: Partial Syntax of a TRA specification in *CLEOPATRA*

```

TRA-class integrate(double TICK, TICK_ERROR)
    in(double) -> out(double)
{
    state:
        double x0 = 0, x1 = 0, y = 0;
    act:
        in(x1) -> :
        ;
        init(),out() -> out(y):
            within [TICK-TICK_ERROR~TICK+TICK_ERROR]
            commit { y = y+TICK*(x0+x1)/2; x0 = x1; }
}

```

Figure 3: Specification of the class of integrators that use the trapezoidal rule.

TRA classes are parametrized. For instance, the specification of `integrate` given in Figure 4 includes the parameters `TICK`, and `TICK_ERROR`, which have to be specified before *instantiating* an object from that class.

The header of a **TRA** class determines its external signature and signaling range function. For example, any **TRA** from the class `integrate` specified in Figure 3 has a signature consisting of an input channel `in` and an output channel `out`. Both `in` and `out` carry actions whose values are drawn from the set of reals. In *CLEOPATRA*, the start channel of any given **TRA**-class is called `init`. Start channels do not have to be explicitly included in the header of a **TRA**-class. For example, in the definition of the `integrate` **TRA**-class given in Figure 3, there is no mention of any `init` channels in the external signature specified in the header, yet, `init` is used later in the body of `integrate`.

The body of a **TRA** class determines the behavior of objects from that class. Such a behavior can be either *basic* or *composite*. The description of a basic behavior involves the specification of a state space in the `state:` section, the specification of an initialization of that space in the `init:` section, and the specification of a set of Time-constrained Event-driven Transactions in the `act:` section. The behavior of an object belonging to the **TRA**-class `integrate` shown in Figure 3 is an example of a basic behavior. Composite behaviors, on the other hand, are specified by composing previously defined, simpler **TRA**-classes together in the `include:` section. For example, in Figure 4, the class `ramp` is defined by composing the `integrate` and `constant`⁸ classes together.

⁸The behavior of an object from the `constant` class is to signal the value `VAL` on its only output channel `out` every $TICK \pm TICK_ERROR$ units of time.

```

TRA-class ramp() -> y(double)
{
  internal:
    x(double) -> ;
  include:
    constant -> x() ;
    integrate x() -> y() ;
}

```

Figure 4: *CLEOPATRA* specification of a ramp generator.

3.2 Time-constrained Event-driven Transaction

In *CLEOPATRA*, the time-constrained causal relationships between events occurring on the different channels of a TRA-class, and the computations (state transitions) that they trigger, are specified using *Time-constrained Event-driven Transactions* (TET). A TET describes the reaction of a TRA to a subset of events. Such a reaction might involve responding to triggers and/or firing action(s). Figure 5 explains the relation between the triggering and firing of actions using TETs.

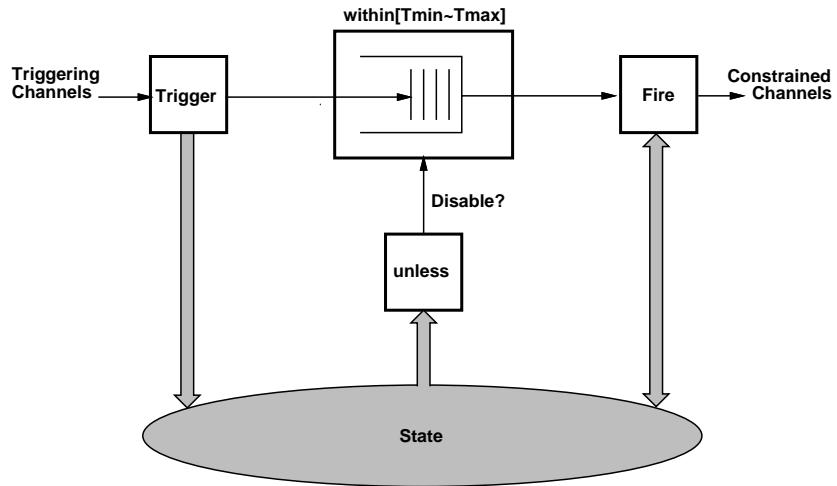


Figure 5: Time-constrained Event-driven Transaction (TET).

The description of a TET consists of two parts: a header and a body. The header of a TET specifies a set of triggering channels (trigger section) and a controlled channel (fire section). The trigger section specifies the effect of the triggering actions on the state of the TRA. In particular, it specifies at most one state variable (per triggering channel) where the value of a trigger on that channel is to be recorded. A TET with no triggering section is triggered every time an action is signaled on any channel of the TRA. In other words, its trigger set is considered to be the same

as the **TRA**'s signature. The fire section specifies the action value to be signaled on the controlled channel as a result of firing the TET. This value can be any expression on the state of the **TRA**. An absent expression means that a random value from the signaling range of the controlled channel is to be signaled. The body of a TET describes possible reactions to the TET triggers. Each reaction is associated with a disabling condition, a time constraint, and a state transformation schema.

For example, the first TET of the **integrate** class shown in Figure 3 is an example of a transaction with only a trigger section. Every time an action is signaled on the input channel **in**, its value is stored in the state variable **x1**, thus, resulting in a potential input transition. The second TET of the **integrate** class, on the other hand, is an example of a transaction with both a trigger section and a fire section. In particular, every time an action is signaled on one of the triggering channels (**init** or **out**) an output action is fired on **out** after a delay of **TICK** \pm **TICK_ERROR** units of time elapses.

Each reaction in the body of a TET is associated with three pieces of information: A disabling condition, a time constraint, and a state transformation schema.

The disabling condition (unless clause) is a boolean expression (predicate) on the state of the **TRA**.⁹ In order to be committed, a reaction's disabling condition has to remain **false** from when the reaction is triggered until it commits. In other words, an intended reaction is aborted if at any point in time after its triggering (scheduling), the disabling condition becomes **true**. The absence of a disabling condition in a reaction implies that, once scheduled, it cannot be disabled.

The time constraint (within clause), determines a lower and upper bound for the real-time delay between scheduling a reaction and committing it. Only constant expressions are allowed to be used in the specification of time bounds. Open, closed, and semi-closed time intervals can be used provided they specify an interval of time from the set \mathcal{D} .¹⁰ The absence of a time constraint from a TET specification implies that the causal relationship between the trigger and its effect is unconstrained in time. A lower bound of 0 and an upper bound of ∞ is assumed in such cases.

The state transformation schema (commit clause) specifies a *method* for computing the next state of the **TRA** once a reaction is committed. We adopt a C-like syntax for the specification of TET methods. Statements in a TET method are executed sequentially. The state transition caused by the execution of a TET method is assumed to be atomic and instantaneous. An absent commit clause implies that committing the reaction does not cause any state changes.

⁹No side effects are permitted in the evaluation of this condition.

¹⁰Current **CLEOPATRA** processors accept only dense intervals of three forms: $(0, T_u)$, (T_l, ∞) , or $[T_l, T_u]$, where $T_u > T_l \geq 0$. These are introduced using the **before**, **after**, and **within** clauses, respectively.

3.3 An Example

Figure 6 shows the specification of a finite FIFO element in *CLEOPATRA*. Values fed into the FIFO element are delayed for some amount of time before being produced as outputs.

```

TRA-class fifo(int N)
    in(float) -> out(float), overflow(), ack()
{
    state:
        float y[N];
        int i, j;
        bool f;
    act:
        init() -> ack():
            before DLY_MIN
            commit { i = 0; j = 0; f = FALSE; }
        in(y[i]) -> ack():
            before DLY_MIN
            commit { i = (i+1)%N ; if (i==j) f = TRUE ; }
        in() -> out(y[j]):
            unless (f)
                within [DLY_MIN~DLY_MAX]
                commit { j = (j+1)%N ; }
        in() -> overflow():
            unless (!f)
                within [DLY_MIN~DLY_MAX]
                ;
}

```

Figure 6: *CLEOPATRA* specification of a finite FIFO delay element.

The header of the **fifo** TRA-class identifies the channel **in** as input, and the channels **out**, **ack** and **overflow** as outputs. Although not explicitly specified as such, the channel **init** (the start channel) is assumed to be an input channel. The signaling range for channels **in** and **out** is the set of floating point numbers, whereas the signaling range for channels **ack** and **overflow** consists of only one value. The body of the **fifo** TRA-class contains two sections. In the **state:** section, the state space of a **fifo** object is described by four state variables: a vector **y[]** of **N** floating point values, two integer values **i** and **j**, and a boolean value **f**. In the **act:** section, the behavior of a **fifo** object is described by four TETs, each of which underscores a causal relationship between the events triggering its execution and those resulting from its execution.¹¹

The first TET in the body of the FIFO establishes a causal relationship between events signaled on **init** and those signaled on **ack**. In particular, firing an action on **init** (the trigger) *causes* the firing of an action on **ack** (the result) after a delay of at most **DLY_MIN**. The

¹¹In other words, between input and output transitions.

second TET establishes a similar causal relationship between events signaled on `in` and `ack`. The third TET establishes a causal relationship between events signaled on `in` and `out`. In particular, firing an action `action` on `in` *causes* the firing of an action on `out` after a delay of at least `DLY_MIN` and at most `DLY_MAX` elapses, provided that the FIFO did not overflow as of the last initialization. The causal relationship that the fourth TET establishes can be explained similarly.

Each TET in a `TRA`-class specifies up to two possible state transitions. Consider, for example, the second TET in the FIFO specification given in Figure 6. In response to a trigger on `in`, the value of the triggering signal is stored in the state variable `y[i]`, thus resulting in a possible state change. Notice that this transition cannot be blocked or delayed; it is an *input transition*. The second state transition, an *output transition*, occurs with the firing of an action on `ack`, resulting in the adjustment of the values of the state variables `i` and `f`. Notice that the value of the action signaled on a local (output or internal) channel does not reflect the state change associated with it. For instance, in the fourth TET of Figure 6, the value signaled on the `out` channel, namely `y[j]`, does not reflect the changes introduced in the `commit` clause, namely advancing the pointer `j`.

3.4 Case and Point!

It is important to realize that `fifo` objects will behave as expected only if inputs from the environment meet certain conditions. In particular, the value of the index `i` is not incremented as a result of an input on the channel `in` until at least `DLY_MIN` units of time elapse following the signaling of that input. It follows that an erroneous behavior will result if two or more events are signaled on the channel `in` in a duration of time shorter than `DLY_MIN`. To avoid such a malignant behavior, the environment must wait for an acknowledgment `ack()`¹² or else, must wait for at least `DLY_MIN` before signaling a new input. Such correctness (safety) conditions can be verified using `TRA`-based verification techniques [Best91c].

We argue that any finite implementation of a `fifo` object (discrete-event delay element) must have a *finite* capacity, which must not be exceeded for a correct behavior. Using *CLEOPATRA*, it is impossible to specify a `fifo` class that behaves correctly *independent* of its environment's behavior. This is a direct result of our abidance by the causality and spontaneity principles, which are preserved by the `TRA` model. As we mentioned at the outset of this paper, it is our thesis that preventing the specification of physically-impossible objects is desired. At the least is spares system developers from trying to implement the impossible.

¹²An `ack()` event is signaled when the previous input has been processed.

4 *CLEOPATRA*: A Simulation Language

We have developed a compiler that transforms *CLEOPATRA* specifications into an event-driven simulator for validation purposes. We have used the *CLEOPATRA* compiler to simulate a variety of systems. In particular, we used it extensively to specify and analyze sensori-motor robotics applications [Best90c] and to simulate complex behaviors of autonomous creatures [Best91a]. Figure 7 shows the different stages involved in the compilation and execution of specifications written in *CLEOPATRA*.

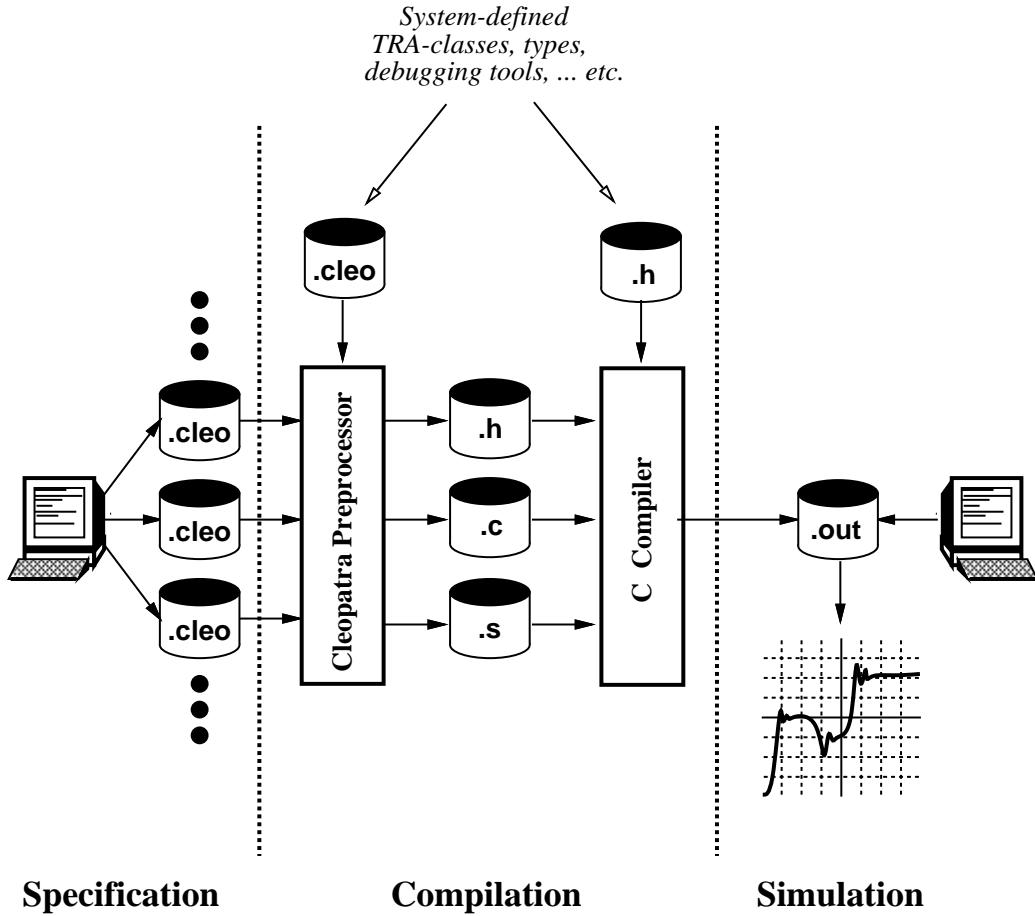


Figure 7: Compilation and simulation of *CLEOPATRA* specifications.

At the heart of this process is a one-pass preprocessor, written in C, which parses user-defined *CLEOPATRA* specifications, augmented with system-defined TRA classes,¹³ and generates an

¹³System-defined TRA classes are mainly for i/o and debugging purposes.

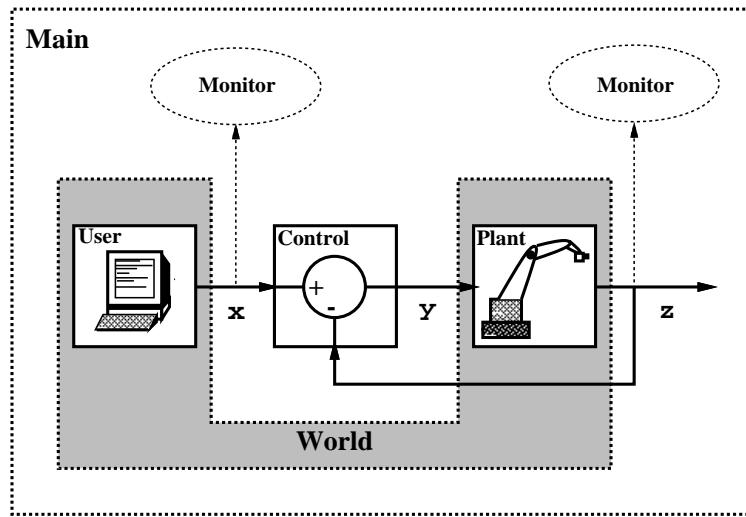


Figure 8: A stand-alone process control system.

```

#include "sysTRA.cleo"

#define TAU 1
#define DLY 5

TRA-class user(double EPOCH)
  -> x(double)
{
  act:
    init(), x() -> x(random(0,1));
    within [0.8*EPOCH^1.2*EPOCH]
    ;
}

TRA-class plant(double GAIN)
  y(double) -> z(double)
{
  state:
    double drive = 0, val = 0 ;

  act:
    y(drive) -> :
    ;
    init(), z() -> z(val);
    within [0.9*DLY^1.1*DLY]
    commit {
      val = val + GAIN*drive ;
    }
}

TRA-class world()
  y(double) -> x(double), z(double)
{
  include:
    user(300) -> x() ;
    plant(1.5) y() -> z() ;
}

TRA-class control()
  x(double), z(double) -> y(double)
{
  state:
    double s = 0, f = 0;

  act:
    x(s), z(f) -> y(s-f):
    within [0.95*TAU^1.05*TAU]
    ;
}

TRA-class main() ->
{
  internal:
    -> x(double), y(double), z(double)
  include:
    world y() -> x(), z() ;
    control x(), z() -> y() ;
    fmonitor("x.dat") x() -> ;
    fmonitor("z.dat") z() -> ;
}

```

Figure 9: The main TRA-class.

equivalent C simulator. This C simulator consists of three components. The first is a header (`.h`) file, which includes type definitions for the state space of the various **TRA** classes in the specification. The second is a schema (`.s`) file, which includes definitions for the state transition functions of the various TETs. The third is the code (`.c`) file, which includes the simulator initialization and control structure along with the instantiation code for the various **TRA** classes, including `main`. The final step of this process involves the invocation of the C compiler to produce an executable simulator. Figure 10 illustrates a typical session, in which the *CLEOPATRA* compiler `ccleo` is invoked to process the file `process-ctrl.cleo` containing the specification of the stand-alone process control system shown in Figures 8 and 9.

In *CLEOPATRA*, any **TRA**-class with no input channels represents a stand-alone (closed) system whose behavior is independent from the outside world; it is a world of its own. One such **TRA**-class, namely `main`, is singled out by *CLEOPATRA* to represent the entire system being specified. For embedded systems, a typical `main` **TRA**-class will simply be the composition of a programmed system, representing the control system, and an external interface, representing the environment. For example, the `main` **TRA**-class shown in Figure 9 represents the *CLEOPATRA* specification of the closed process control system shown in Figure 8. The execution of a *CLEOPATRA* stand-alone system is started by instantiating an object from the **TRA**-class `main` at time¹⁴ 0 and, thereafter, committing only the legal transitions dictated by the system specification and the semantics of the **TRA** model. Figure 11 shows the values signaled on the `x` and `z` channels over time.

A library of system-defined **TRA**-classes is available for debugging and performing I/O in *CLEOPATRA*. For example, in the specification of the **TRA**-class `main` given in Figure 9, the **TRA**-class `fmonitor` is used to record the action values signaled on the `x` and `z` channels in files `x.dat` and `z.dat` respectively. System-defined **TRA**-classes are themselves specified in *CLEOPATRA*. They are different from user-defined **TRA**-classes in that they have access to *global* information known only to the simulator. For instance, `fmonitor` objects have access to the simulator's *perfect* clock, `_clk`, whereas user-defined **TRA**-classes have to maintain their own locally *perceived* clocks, if needed.

C functions can be called from within a *CLEOPATRA* specification. To maintain the semantics of the **TRA** formalism, however, only functions with no side effects should be used. In other words, C function should be restricted to act as pure operations on the state variables of an object. It should not reach beyond the boundaries of the state space of that object. Also, it should not alter the structure of the state space of the object in any way. An example of the use of a C-function is illustrated in the description of the `user` **TRA**-class of Figure 9 where the function `random()` is called periodically to generate a random set value.

¹⁴The start time of the simulation can be explicitly specified.

```

% ccleo process-ctrl
TRA-class fmonitor(string FILENAME)
  init(unit), signal(double) -> ;
TRA-class user(double EPOCH)
  init(unit) -> x(double) ;
TRA-class plant(double GAIN)
  init(unit), y(double) -> z(double) ;
TRA-class world()
  init(unit), y(double) -> x(double), z(double) ;
TRA-class control()
  init(unit), x(double), z(double) -> y(double) ;
TRA-class main()
  init(unit) -> 'z(double)', 'y(double)', 'x(double)' ;

Cleopatra preprocessing completed.
C compilation completed.

% process-ctrl
CPU time = 1366612 usec  # of events = 5486    SEPS = 4014.3069

```

Figure 10: A typical *CLEOPATRA* compilation and execution session.

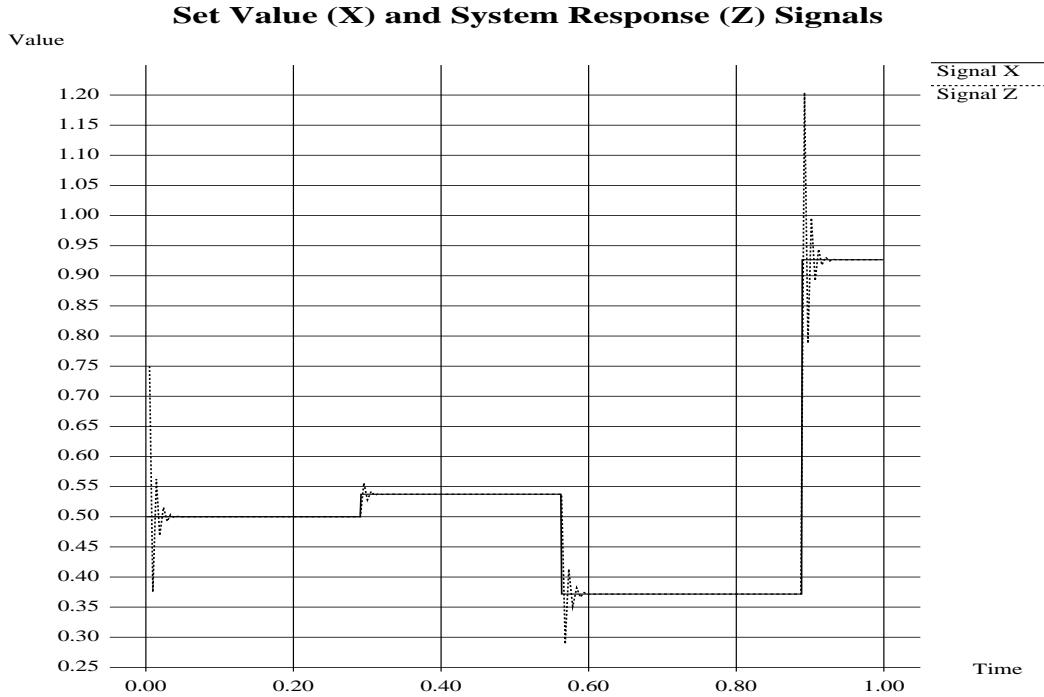


Figure 11: Simulated behavior of an underdamped process control system.

Most of the C preprocessor utilities are available in *CLEOPATRA*. This includes simple and parameterized macro definition and invocation, constant definition, and nested file inclusion.¹⁵ For example, in the *CLEOPATRA* specification of the stand-alone process control system shown in Figure 9, system-defined TRA classes are included using the `#include` directive, and constants are defined using the `#define` directive.

The simulator has proven to be quite efficient. This is due primarily to the causal and compositional nature of the TRA model, which tend to localize the computation triggered by the occurrence of an event within the boundaries of few TETs. The number of simulated events per second (seps) depends on a number of factors: the average channel fan-out, the average number of TETs per TRA, and the complexity of the event-driven computation. It does not depend, however, on the size of the state space or on the amount of TRA nesting. For an application with a fan-out of 1 and an average of 2.4 TETs per TRA, and an $O(1)$ event-driven computational complexity, the compiled *CLEOPATRA* specifications executed at a rate of almost 19,500 seps.¹⁶ The performance of a simulator for the same application hand coded directly in C performed only slightly better. Namely, it executed at a rate of almost 20,000 seps. The performance of the simulator degrades considerably when extensive I/O and tracing operations are performed.¹⁷

5 Conclusion

Predictability can be *enhanced* in a variety of ways. It can be enhanced by restricting expressiveness as was done in Real-Time Euclid [Klig86], by sacrificing accuracy as was done in the Flex system [Chun90], or by abstracting segmented resources as was done in the Spring kernel [Stan89]. The TRA-development methodology we are advocating here introduces one more way of improving predictability, that of allowing only physically-sound specifications. Pursuing the ideas presented in this paper will undoubtedly provide us with one more handle in our persistent quest for predictable systems. An interesting question to be addressed in the future would be whether this and other handles can be combined in any useful way to *guarantee* predictability.

Our experience with the TRA development methodology in the design, simulation, and analysis of asynchronous digital circuits, sensori-motor autonomous systems, and intelligent controllers confirms its suitability for the specification, verification, and validation of many embedded and time-critical applications. Its usefulness in the implementation of such systems, although promising, is

¹⁵Current *CLEOPATRA* processors do not admit conditional compilation.

¹⁶All simulations were performed on a SPARCstation SLC™ workstation.

¹⁷This is the case in the simulation shown in Figure 10, where an almost 5-fold decrease in efficiency can be attributed to the use of the `fmonitor` TRA-class.

yet to be established. An fruitful direction for future research would be to automate the process of transforming **TRA**-based physically-sound time-critical specifications into provably-correct implementations given appropriate resources. Such research will have two complementary – experimental and theoretical – components. The experimental component would involve the development of a compiler to transform **CLEOPATRA** specifications into predictable real-time programs, given a dedicated computing platform. The theoretical component would aim at devising efficient verification algorithms that can be automated and incorporated in the **CLEOPATRA** compiler.

References

- [Alur90] Rajeev Alur, Costas Courcoubetis, and David Dill. “Model-checking for real-time systems.” In *Proceedings of the 5th annual IEEE Symposium on Logic in Computer Science*, Philadelphia, Pennsylvania, June 1990. IEEE Computer Society Press.
- [Best90a] Azer Bestavros. “The IOTA: A model for real-time parallel computation.” In *Proceedings of TAU'90: The 1990 ACM International Workshop on Timing issues in the Specification and Synthesis of Digital Systems*, Vancouver, Canada, August 1990.
- [Best90b] Azer Bestavros. “TRA-based real-time executable specification using CLEOPATRA.” In *Proceedings of the 10th Annual Rochester Forth Conference on Embedded Systems*, Rochester, NY, June 1990. (revised May 1991).
- [Best90c] Azer Bestavros, James Clark, and Nicola Ferrier. “Management of sensori-motor activity in mobile robots.” In *Proceedings of the 1990 IEEE International Conference on Robotics & Automation*, Cincinnati, Ohio, May 1990. IEEE Computer Society Press.
- [Best91a] Azer Bestavros. “Planning for embedded systems: A real-time prospective.” In *Proceedings of AIRTAC-91: The 3rd IFAC Workshop on Artificial Intelligence in Real Time Control*, Napa/Sonoma Region, CA, September 1991.
- [Best91b] Azer Bestavros. “Specification and verification of real-time embedded systems using the Time-constrained Reactive Automata.” In *Proceedings of the 12th IEEE Real-time Systems Symposium*, pages 244–253, San Antonio, Texas, December 1991.
- [Best91c] Azer Bestavros. *Time-constrained Reactive Automata: A novel development methodology for embedded real-time systems*. PhD thesis, Harvard University, Division of Applied Sciences (Department of Computer Science), Cambridge, Massachusetts, September 1991.
- [Best92] Azer Bestavros, Devora Reich, and Robert Popp. “Cleopatra compiler design and implementation.” Technical Report TR-92-019, Computer Science Department, Boston University, Boston, MA, August 1992.
- [Burn90] Alan Burns and Andy Wellings. *Real-time systems and their programming languages*. Addison Wesley Co. (International Computer Science Series), 1990.
- [Chun90] Jen-Yao Chung, Jane Liu, and Kwei-Jay Lin. “Scheduling periodic jobs that allow imprecise results.” *IEEE Transaction on Computers*, 19(9):1156–1173, September 1990.
- [Clar91] James Clark, Nicola Ferrier, and Lei Wang. “A robotics system for manipulation using directed vision feedback.” Internal report, Robotics laboratory, Harvard University, Cambridge, MA, 1991.
- [Fu87] K. S. Fu, R. C. Gonzalez, and C. S. G. Lee. *Robotics: Control, sensing, vision, and intelligence*. McGraw-Hill Book Company, 1987.
- [Hawk88] Stephen W. Hawking. *A brief history of Time: From the Big Bang to Black Holes*. Bantam Books, April 1988.
- [Klig86] Eugene Kligerman and Alexander Stoyenko. “Real-time Euclid: A language for reliable real-time systems.” *IEEE Transactions on Software Engineering*, 12(9):941–949, September 1986.

[Lewi90] Harry Lewis. “A logic of concrete time intervals.” In *Proceedings of the 5th annual IEEE Symposium on Logic in Computer Science*, Philadelphia, PA, June 1990. IEEE Computer Society Press.

[Lync88a] Nancy Lynch and Kenneth Goldman. “6.852 distributed algorithms lecture notes: The I/O Automata.” Technical report, Laboratory of Computer Science, MIT, Cambridge, MA, Fall 1988.

[Lync88b] Nancy Lynch and Mark Tuttle. “An introduction to Input/Output Automata.” Technical Report MIT/LCS/TM-373, MIT, Cambridge, Massachusetts, November 1988.

[Lync91] Nancy Lynch and Frits Vaandrager. “Forward and backward simulations for timing-based systems.” Unpublished notes, Massachusetts Institute of Technology Laboratory for Computer Science, August 1991.

[Lyon89] Damian Lyons and Michael Arbib. “A formal model of computation for sensory-based robotics.” *IEEE Transactions on Robotics and Automation*, 5(3):280–293, 1989.

[Schn88] Fred Schneider. “Critical (of) issues in real-time systems: A position paper.” Technical Report 88-914, Department of Computer Science, Cornell University, Ithaca, NY, May 1988.

[Sree90] Ramavarapu Sreenivas. *Towards a system theory for interconnected Condition/Event systems*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, September 1990.

[Stan88a] John Stankovic. “Misconceptions about real-time computing.” *IEEE Computer*, October 1988.

[Stan88b] John Stankovic and Krithi Ramamritham, editors. *Hard Real-Time Systems*. IEEE Computer Society Press, 1988.

[Stan89] John Stankovic and Krithi Ramamritham. “The Spring Kernel: A new paradigm for real-time operating systems.” *ACM Operating Systems Review*, 23(3):54–71, July 1989.

[Stua91] D.A. Stuart and P.C. Clements. “Clairvoyance, capricious timing faults, causality, and real-time specifications.” In *Proceedings of the 12th IEEE Real-time Systems Symposium*, pages 254–263, San Antonio, Texas, December 1991.

[Tilb91a] André M. van Tilborg and Gary M. Koob, editors. *Foundations of Real-Time Computing: Formal Specifications and Methods*. Kluwer Academic Publishers, 1991.

[Tilb91b] André M. van Tilborg and Gary M. Koob, editors. *Foundations of Real-Time Computing: Scheduling and resource management*. Kluwer Academic Publishers, 1991.

[Tutt88] Mark Tuttle, Michael Meritt, and Francesmary Modugno. “Time constrained automata.” MIT/LCS, November 1988.

[Wirt77] Niklaus Wirth. “Toward a discipline of real-time programming.” *Communications of the ACM*, 20(8), August 1977.